Minimization Algorithms for Multiple-Valued Programmable Logic Arrays

Parthasarathy P. Tirumalai, *Member, IEEE,* and Jon T. Butler, *Fellow, IEEE*

***Abstract*—We analyze the performance of various heuristic algorithms for minimizing realizations of multiple-valued functions by the newly developed CCD [9] and CMOS [15] programmable logic arrays. The functions realized by such PLA's are in sum-of-products form, where sum is ordinary addition truncated to the highest logic value, and where product represents the MIN operation on functions of the input variables which are the interval literal operations. We compare three previously published heuristics, Pomper and Armstrong [14], Besslich [3], and Dueck and Miller [6], over sets of random and random symmetric functions. We show an exact minimization method that is a tree search using backtracking. A considerable reduction in the search space is achieved by considering *constrained implicant sets* and by eliminating some implicants altogether. Even with this improvement, the time required for exact minimization is extremely high when compared to all three heuristics. We also examine the case where only *prime implicants* are considered and show that such implicants have marginal value compared to constrained implicant sets. Our basis of comparison is the average number of product terms. We show that the heuristic methods are reasonably close to minimal and produce nearly the same average number of product terms. Interestingly though, there is surprisingly little overlap in the set of functions where the best realization is achieved. Thus, there is a benefit to applying all three heuristics to a given function and then choosing the best realization.**

***Index Terms*— Absolute minimization, heuristic minimization, implicant, multiple-valued logic, prime implicant, programmable logic arrays, sum-of-products.**

I. Introduction

THE minimization of sum-of-products expressions in binary logic has received considerable attention for over 30 years. The complexity of the problem has been known for almost as long. In 1965, Gimpel [7] showed that any instance of the set covering problem is an instance of sum-of-products extraction. In 1972, Karp [8] showed that the set covering problem is NP complete; thus, so also is sum-of-products extraction. The best known algorithm then requires exponential time. This is a real barrier; it precludes the exact minimization of functions with even a moderately low number of inputs, e.g., 20. As a result, considerable effort has been devoted to heuristic minimization methods. For example, among the Berkeley VLSI tools is ESPRESSO-IIC [4], a C program that minimizes binary functions by a set of operations on the prime implicants.

Recently, there has been considerable interest in multiple-valued PLA's [1]–[3], [5], [6], [9]–[12], [14]–[19]. Several have been proposed [12], [15], [17], [18] and at least one implemented [9]. We know of three heuristic multiple-valued sum-of-products minimization algorithms. All three use the direct cover method, which proceeds in two steps: 1) select a minterm, and 2) select an implicant. Pomper and Armstrong [14] introduced in 1981 a direct cover method that finds a near-minimal sum-of-products expression by choosing a random minterm and an implicant covering that minterm which, when subtracted, drives the most number of minterms to 0. In 1986, Besslich [3] introduced another direct cover method that seeks to cover the "most isolated" minterms first. And in 1987, Dueck and Miller [6] introduced a method which also seeks the most isolated minterm first, but chooses a product term that tends to introduce the fewest discontinuities when subtracted from the function.

There has been little study of the relative merits of available heuristic algorithms even in binary logic. To the credit of Brayton, Hachtel, McMullen, and Sangiovanni-Vincentelli [4], the realizations produced by ESPRESSO-IIC were compared to the realizations of MINI, PRESTO, and POP [4] over a set of 56 specifically chosen binary functions. To our knowledge, there has been no comparative analysis of minimization algorithms for multiple-valued logic. The justification of the newly introduced heuristics examined by us has rested on an intuitive notion, supported by examples. In this paper, we analyze the following synthesis methods over sets of 7000 random and 7000 random symmetric 4-valued 2-variable functions.

1) Random-minterm/random-implicant
2) Pomper and Armstrong [14]
3) Besslich [3]
4) Dueck and Miller [6]
5) Gold
6) Absolute minimization using prime implicants
7) Absolute minimization using all implicants.

Our results show the Besslich heuristic is slightly inferior to the Dueck and Miller heuristic while there is superiority of each over the Pomper and Armstrong heuristic. Our analysis is based on how well each heuristic minimizes functions from a sample set of randomly chosen 4-valued 2-variable functions. We impose a uniformly distributed probability distribution. Thus, we avoid the bias associated with a specifically chosen set. Since *symmetric* functions occur commonly in practice, we perform a similar analysis over a set of randomly chosen 4-valued 2-variable symmetric functions. Again, a uniform probability distribution is imposed.

Our choice of 2-variable functions is influenced by a need to compare heuristics (1–5) to minimal realizations (6–7). Even with only 2-inputs, it requires a day to compute minimal realizations. Our choice of 4-values is influenced by the current interest in this radix. Also, since the PLA's produced in CCD [9] and CMOS [15] use the truncated SUM on interval literals of the input variables, in our analysis we use these operations. We have adapted the three heuristic algorithms in [3], [6], [14] to this case. The heuristics in [6] and [14] were modified slightly to conform to our study. A motivation for the study reported here is the development of a minimization method for a CCD PLA CAD tool [10].

Manuscript received September 29, 1988; revised April 13, 1989. This work was supported in part by NSF under Grant MIP-8706553, NATO under Grant 423/84, the Naval Research Laboratory, the Naval Postgraduate School, and a Hewlett-Packard Fellowship. A preliminary version of this paper was presented at the 18th International Symposium on Multiple-Valued Logic, Palma de Mallorca, Spain, May 1988.

P. P. Tirumalai is with Hewlett-Packard Laboratories, Palo Alto, CA 94304.
J. T. Butler is with the Naval Postgraduate School, Monterey, CA 93943.

IEEE Log Number 9041279.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
APR 1989		

4. TITLE AND SUBTITLE
Minimization Algorithms for Multiple-Valued Programmable Logic Arrays

5a. CONTRACT NUMBER
5b. GRANT NUMBER
5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

5d. PROJECT NUMBER
5e. TASK NUMBER
5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School,Department of Electrical and Computer Engineering,Monterey,CA,93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)
11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

We analyze the performance of various heuristic algorithms for minimizing realizations of multiple-valued functions by the newly developed CCD 191 and CMOS [W] programmable logic arrays. The functions realized by such PLA?s are in sum-of-products form, where sum is ordinary addition truncated to the highest logic value, and where product represents the MIN operation on functions of the input variables which are the interval literal operations. We compare three previously published heuristics, Pomper and Armstrong [14], Besslich [3], and Dueck and Miller [6], over sets of random and random symmetric functions. We show an exact minimization method that is a tree search using backtracking. A considerable reduction in the search space is achieved by considering constrained implicunt sets and by eliminating some implicants altogether. Even with this improvement, the time required for exact minimization is extremely high when compared to all three heuristics. We also examine the case where only prime implicunfs are considered and show that such implicants have marginal value compared to constrained implicant sets. Our basis of comparison is the average number of product terms. We show that the heuristic methods are reasonably close to minimal and produce nearly the same average number of product terms. Interestingly though, there is surprisingly little overlap in the set of functions where the best realization is achieved. Thus, there is a benefit to applying all three heuristics to a given function and then choosing the best realization.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**11**	

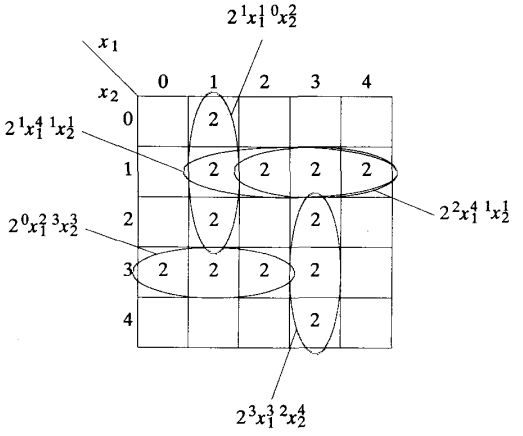


Fig. 1. Example of a 5-valued 2-variable function.

This paper is organized as follows. The next section introduces notation and fundamental concepts. Section III describes the seven synthesis methods, and Section IV describes the results of a comparative analysis of their performance. Section V describes the absolute minimization algorithm. In the final section, we summarize the results.

II. Background and Notation

Let $\boldsymbol{X} = \{x_1, x_2, \cdots, x_n\}$ be a set of n variables, where x_i takes on values from $\boldsymbol{R} = \{0, 1, \cdots, r-1\}$. A function $f(\boldsymbol{X})$ is a mapping $f : \boldsymbol{R}^n \rightarrow \boldsymbol{R} \cup \{r\}$, where r is the *don't care* value. Specifically, $f(\boldsymbol{X})$ is said to be an *r-valued n-variable function.* Fig. 1 shows a map representation of a 5-valued 2-variable function. Blank entries correspond to 0. An assignment $\boldsymbol{x}$ of values to variables in $\boldsymbol{X}$ is called a *minterm* if $f(\boldsymbol{x}) \neq 0$. In Fig. 1, there are 12 minterms all of which yield a 2.

Functions realized by the CCD [9] and CMOS [11], [15] PLA's are composed of three operations:

1) MIN : $f(x_1, x_2) = x_1 x_2 \quad (= \text{MIN}(x_1, x_2))$,
2) SUM : $f(x_1, x_2) = x_1 + x_2 \quad (= x_1 + x_2$ if $x_1 + x_2 \leq r-1$ and $= r-1$ otherwise), where x_i is viewed as an integer and + is viewed as integer addition, and
3) literal: $f(x_1) = {}^a x_1^b \quad (= r-1$ if $a \leq x_1 \leq b$ and $=$ 0, otherwise).

We use only 4-valued functions in this paper. Thus, $r = 4$. In binary, the SUM, MIN, and literal correspond to AND, OR, and $\boldsymbol{x}^*$, where $\boldsymbol{x}^* \in \{x, \overline{x}, 1\}$. In the realization of functions by a multiple-valued PLA, both constants and literals occur as operands of the MIN functions. A *product term* is the MIN of one nonzero constant and one or more literals. For example, $f(x_1, x_2) = 2^2 x_1^{4\ 1} x_2^1$ is a product term that is 2 when x_1 is 2, 3, or 4 and x_2 is 1. An *implicant* of a function $f(\boldsymbol{X})$ is a product term $I(\boldsymbol{X})$ such that $f(\boldsymbol{X}) \geq I(\boldsymbol{X})$ where $\geq$ means that for *all* assignments $\boldsymbol{x}$ of values to $\boldsymbol{X}$, $f(\boldsymbol{x}) \geq I(\boldsymbol{x})$. Also, $|I(\boldsymbol{X})|$ denotes the nonzero constant associated with $I(\boldsymbol{X})$. A *prime implicant* of $f(\boldsymbol{X})$ is an implicant of $f(\boldsymbol{X})$ such that there is no other implicant $I'(\boldsymbol{X})$ of $f(\boldsymbol{X})$ where $I'(\boldsymbol{X}) \geq I(\boldsymbol{X})$. For example, $2^2 x_1^{41} x_2^1$ is an implicant of $f(x_1, x_2)$, but it is not a prime implicant. However, $2^1 x_1^{4\ 1} x_2^1$ is a prime implicant. Any function can be expressed as the SUM of implicants [18]. For example, the function in Fig. 1 can be expressed as the SUM of 4 implicants,

$$f(x_1, x_2) = 2^2 x_1^{4\ 1} x_2^1 + 2^3 x_1^{3\ 2} x_2^4 + 2^0 x_1^{2\ 3} x_2^3 + 2^1 x_1^{1\ 0} x_2^2. \quad (1)$$

We use the term *sum-of-products* to describe functions realized by such PLA's, where sum refers to SUM and product refers to MIN. A sum-of-products expression for function $f(\boldsymbol{X})$ is *minimal* if there is no other expression for $f(\boldsymbol{X})$ with fewer product terms. For example, (1) is a minimal sum-of-products expression. Given $f(\boldsymbol{X})$, implicant $I(\boldsymbol{X})$ *covers* a minterm at $\boldsymbol{x}$ if $f(\boldsymbol{x}) = I(\boldsymbol{x})$. Therefore, $g(\boldsymbol{X}) = f(\boldsymbol{X}) - I(\boldsymbol{X})$ has the property $g(\boldsymbol{x}) = 0$, and we say that subtracting $I(\boldsymbol{X})$ *drives* the minterm at $\boldsymbol{x}$ to 0.

It is interesting to note that *none* of the implicants in (1) are prime. In fact, any sum-of-products expression for $f(x_1, x_2)$ containing at least one prime implicant is *not* minimal. Such expressions require at least 5 product terms. Indeed, there is no sum-of-products expression for $f(\boldsymbol{X})$ containing only its prime implicants. Therefore, in contrast to the case of conventional binary sum-of-products, it is not sufficient to consider just prime implicants [19]. A similar observation was made with respect to sum-of-products expressions, where sum is MAX and products are the MIN of unary functions of varying costs [13].

III. Minimization Algorithms for SUM PLA's

We consider seven synthesis methods. The first five are based on the direct cover approach. In this approach, a minterm is first chosen. Next, an implicant is chosen which covers this minterm. The implicant is then subtracted from the function and the process is repeated until there are no more minterms.

A. Random-Minterm/Random-Implicant

In this method, both the minterm and the implicant are chosen randomly, with all choices being equally likely. That is, at each stage of the covering process, one minterm is chosen randomly from among all candidate minterms, and similarly in the case of implicants. As with the other algorithms, the chosen implicant is subtracted and the process repeated on the resulting function. Since no particular characteristics of the function are used to determine the choice of minterm or implicant, it provides a basis of comparison for the next four heuristics, which use characteristics of the functions to limit, in varying degrees, the choices of minterms and implicants.

B. Pomper and Armstrong

In our adaptation of this heuristic, the minterm is chosen randomly just as in the random-minterm/random-implicant case. However, the implicant is chosen as the one, which when subtracted, drives the largest number of minterms to 0 or don't care. If there is more than one such implicant, the largest is chosen. If there is more than one largest, the one generated first is chosen. The process is repeated until the function is completely covered. A formal description of this algorithm is given in Appendix I.

The random nature of these two methods does not allow repeatable results. Because the random-minterm/random-implicant method can choose from all possible covers of any function, there is a nonzero probability that it will find a minimal realization. However, if minimal realizations are a small percentage of the total number of realizations, nonminimal realizations will most likely result. Our analysis shows this clearly. There is then the question of whether the Pomper and Armstrong heuristic has a nonzero probability of finding a minimal realization for

all functions. The example in Fig. 1 is a function where this probability is 0. Here, an implicant which when subtracted drives the most number of minterms to 0 is never in a minimal sum-of-products realization, and so, for this case, this heuristic *cannot* produce a minimal realization.

C. Besslich

Besslich [3] presents a direct cover approach for the heuristic minimization of multiple-valued logic functions using minterm weighting and implicant detecting transformations. In this heuristic, each minterm is assigned a weight that is a measure of the degree to which other minterms cluster around it. Minterms in the center of clusters have the highest weight and isolated minterms the lowest. The minterm with the smallest weight is chosen. Next, all implicants that cover the chosen minterm are generated. For each, an efficiency factor equal to the number of minterms it covers divided by its cost is calculated, and the minterm with the largest factor is chosen. In a PLA, all implicants are realized with the same cost (one column). Therefore, for this case, only the number of minterms driven to 0 or don't care determines which implicant is chosen. The basic idea of the Besslich heuristic is to cover the most isolated minterms first and use implicants that have a low cost per minterm covered. Thus, the choice of implicants is made in the same way as in the Pomper and Armstrong heuristic. Besslich [3] does not mention how ties are broken. We choose to break ties among implicants having the same efficiency factor by using only the largest implicants, and among these by choosing the first one generated. A formal description of the heuristic is given in Appendix II.

D. Dueck and Miller

Dueck and Miller [6] present a heuristic similar to that of Besslich's with the intent of improving realizations using the SUM operation. In this approach, an isolation factor (IF) is calculated for each minterm with the smallest value (that is, all 1 minterms are considered first; if there are none, then 2 minterms are considered, etc.). The isolation factor of a minterm is inversely proportional to 1 plus the number of *adjacent* minterms plus the number of logic variables where there are a nonzero number of such minterms. Similar to the weight transform presented by Besslich, the isolation factor provides a measure of the degree to which a specific minterm can combine with other minterms in the function. The minterm with the highest isolation factor is chosen. All implicants that cover this minterm are then generated. For each, a measure called the relative break count (RBC) is calculated. This provides a measure of the degree to which the function is simplified if the implicant under consideration is chosen. The idea is to judiciously choose implicants that make the remaining function easy to realize. Our use of interval literals requires an adjustment to the Dueck and Miller algorithm. A formal description of this heuristic, modified to the specifications of our problem, is given in Appendix III.

E. Gold

Gold is a heuristic in which the heuristic algorithms of Pomper and Armstrong, Besslich, and Dueck and Miller are applied and the best realization is chosen. It was inspired by the observation that these algorithms displayed a diversity in realizations. That is, no single algorithm is consistently better than the others over all functions; there are classes of functions where one heuristic does better than the others.

TABLE I
SUMMARY OF MINIMIZATION ALGORITHMS

Algorithm	Choice of Minterm	Choice of Implicant
1. Random-Min./Random-Impli.	Random	Random
2. Pomper and Armstrong [14]	Random	Drives most min. to 0
3. Besslich [3]	Smallest weight	Drives most min. to 0
4. Dueck and Miller [6]	Largest IF	Smallest RBC
5. Gold	Best of 2, 3, and 4	
6. Absolute minimization using only prime implicants	Exhaustive search on *prime* implicants	
7. Absolute minimization using all implicants	Exhaustive search on *all* implicants	

F. Absolute Minimization Using Prime Implicants

This algorithm produces the smallest sum-of-products realization but with the restriction that only *prime* implicants of the current function are used at each stage in the covering process. Our intent was to compare this to the case of absolute minimization using all implicants. This algorithm requires considerable computer time. It is described in Section VI.

G. Absolute Minimization Using All Implicants

An algorithm which produces the exact minimal sum-of-products was devised to compare to the results produced by the previous six algorithms. This does essentially an exhaustive search over all possible solutions, starting with the fewest number of product terms. The algorithm also requires considerable computer time. A complete description of the algorithm is given in Section VI.

H. Summary

Table I summarizes the seven algorithms.

IV. Comparison of Algorithm Performance

For the purpose of comparison, we separate all 4-valued, 2-variable functions into 17 disjoint classes according to the number of nonzero values in the function. For 14 of the 17 classes, 500 random and 500 random symmetric functions are generated. For each function, the seven algorithms are applied and the number of product terms derived. For functions with less than three nonzero values, exact values for the average number of product terms can be calculated as follows.

0 nonzero values: There is only one (trivial) function, constant zero, in this class. This function is symmetric, requires no implicants in its cover, and all algorithms produce identical results.

1 nonzero value: There are 48 functions in this class of which 12 are symmetric. All require one implicant. Thus, the average number of implicants required by all algorithms is 1.

2 nonzero values: There are $3^2 \cdot \binom{16}{2} = 1080$ such functions of which only 72 are symmetric. All symmetric functions require two implicants and all algorithms find the minimal cover. Hence, the average for symmetric functions in this class is 2 for all algorithms. Of the 1080 functions in this class, 72 can be covered by one implicant. Only the random-minterm/random-implicant algorithm fails to find the minimal cover for all functions in this class. Specifically, functions with adjacent identical minterms will be incorrectly minimized half the time, since a cover of a single minterm or two adjacent minterms are equally likely. Over the set of 1080 functions with 2 nonzero values, the average

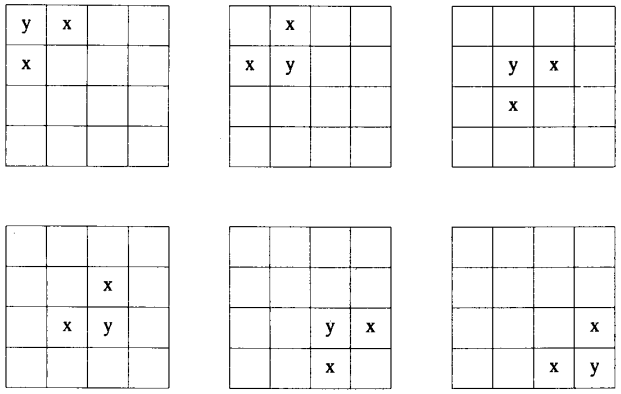


Fig. 2. Patterns of symmetric functions with 3 nonzero values which could be covered with 2 product terms.

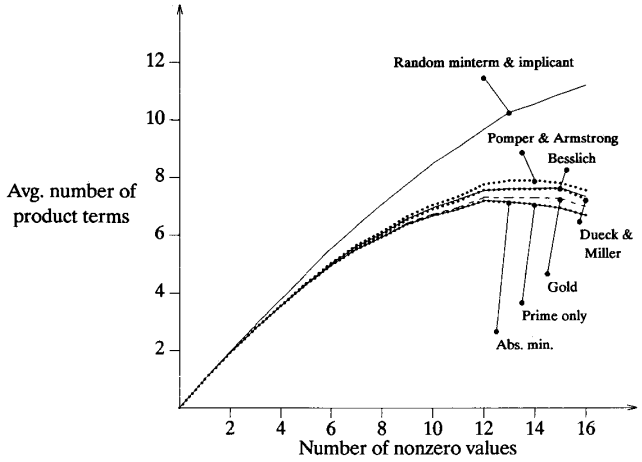


Fig. 3. The average number of product terms versus the number of nonzero values for random functions.

and standard deviation on the number of implicants produced by the random-minterm/random-implicant algorithm are calculated as 1.97 and 0.18, respectively. On the other hand, these values for all other algorithms are calculated as 1.93 and 0.25, respectively. Our random function generation program was tested on this class and yielded values very close to these, to within 0.18% for the average and to within 3.9% for the standard deviation.

For symmetric functions with 3 nonzero values, the exact value of the average number of product terms can be calculated as follows.

3 nonzero values: There are 324 symmetric functions of which 108 have all the nonzero values on the diagonal and therefore need 3 product terms. The remaining 216 functions have one nonzero value on the diagonal. These functions require either 2 or 3 product terms. The functions that can be covered with 2 product terms must have one of the six patterns shown in Fig. 2. Here x & y can be, respectively, 1 & 1, 2 & 2, 3 & 3, 1 & 2, or 2 & 3. Thus, there are a total of $6 \times 5 = 30$ functions that can be covered with 2 product terms. The average and the standard deviation are therefore 2.907 and 0.294, respectively. Our random function generation program was tested on this class and yielded values within 0.35% for the average and to within 4.1% for the standard deviation.

We were not able to obtain exact values for the remaining classes, and we investigated these using sample sets. For each class, 500 functions were randomly generated and all algorithms applied to each. The functions generated had no don't care values. However, they could develop don't care values during the covering process. (Nota bene: A value of 3 in the original function becomes a don't care when fully covered. The covering process stops when only 0's and don't cares remain in the function.) Figs. 3 and 4 show the results.

It can be seen that the relative performance of the various algorithms over the two sets of functions, random and random symmetric, is essentially the same. Thus, the following observations apply to algorithm performance over both sets. The random-minterm/random-implicant algorithm does quite poorly. Choosing the implicant that drives the most minterms to 0, as in the Pomper and Armstrong algorithm, provides a significant improvement. There is little difference between the performance of the Besslich and Dueck and Miller algorithms; their curves in Figs. 3 and 4 are almost identical. Both heuristics choose the most isolated minterm first, and this gives some improvement over that of the Pomper and Armstrong algorithm. Gold, taking advantage of the small overlap between the three heuristic algorithms, provides some further improvement. Still further

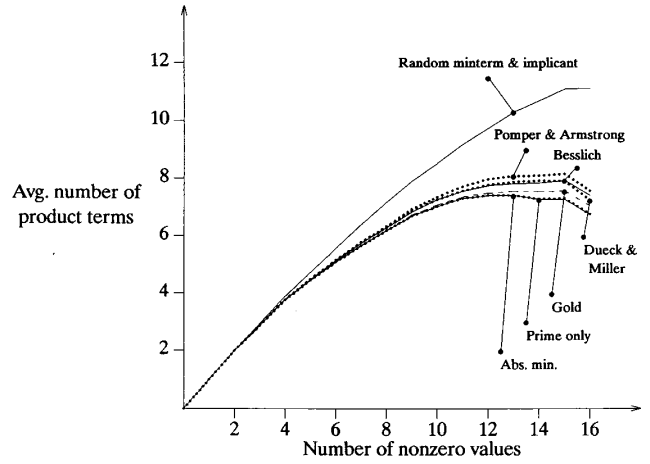


Fig. 4. The average number of product terms versus the number of nonzero values for random symmetric functions.

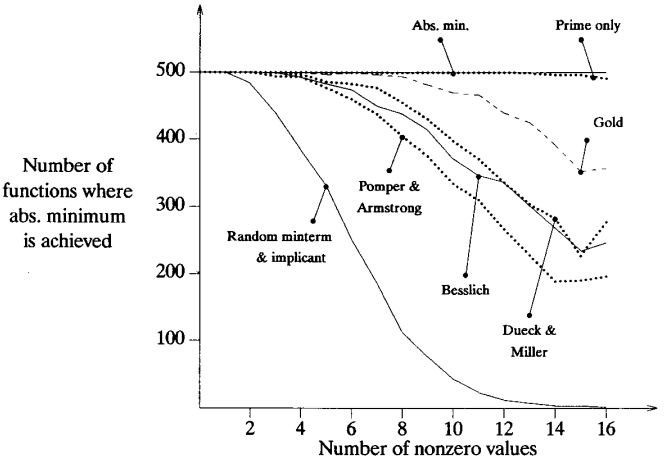


Fig. 5. The number of functions where the absolute minimal realization is achieved versus the number of nonzero values for random functions.

improvement is achieved by the algorithm that obtains the absolute minimum solution using only prime implicants. In fact, the difference between the absolute minimization algorithm using prime implicants and the same when using all implicants is hardly visible in Figs. 3 and 4, although the latter is better.

Another measure of comparison of the algorithms is the *number* of functions for which the absolute minimal solution is found. Figs. 5 and 6 show how the seven algorithms compare on this basis. Here, there is a much larger distinction between the algorithms. As the number of nonzero values increases, the number of functions for which an optimum cover is found by the random-minterm/random-implicant algorithm drops off sharply, to nearly zero. The Pomper and Armstrong algorithm offers considerable improvement, but, even in this case only about 39% (43%) of the functions (symmetric functions) are completely minimized when the number of nonzero values is 14–16. Between the Besslich and the Dueck and Miller algorithms, the latter is slightly better. Approximately, 52% of the functions are minimized for these two algorithms when the number of nonzero values is 14–16. Gold shows a reasonable improvement, minimizing about 75% of the functions with 14–16 nonzero values. Considering only prime implicants yields the minimal solution for about 98% of the functions with 14–16 nonzero values. Again, the algorithms exhibit very similar behavior for both random and random symmetric functions.

The programs for prime implicants only and absolute minimization, and the random function generation routines were written in C. The heuristic algorithms were written in Pascal. The program was executed on a Hewlett-Packard Series 9000 Model 350 workstation running HP-UX (HP's extension of the UNIX™ operating system). The complete program took about one day to minimize 7000 randomly generated symmetric functions with 3–16 nonzero values. Another run was made using different seeds for the random number generator streams. The results from both runs were very similar, indicating that a sufficiently large sample set size was used.

All heuristics took approximately the same time to minimize both sets of 7000 functions. However, the heuristics were not optimized for fast execution; there was an overhead associated with generating the functions and logging results. Furthermore,

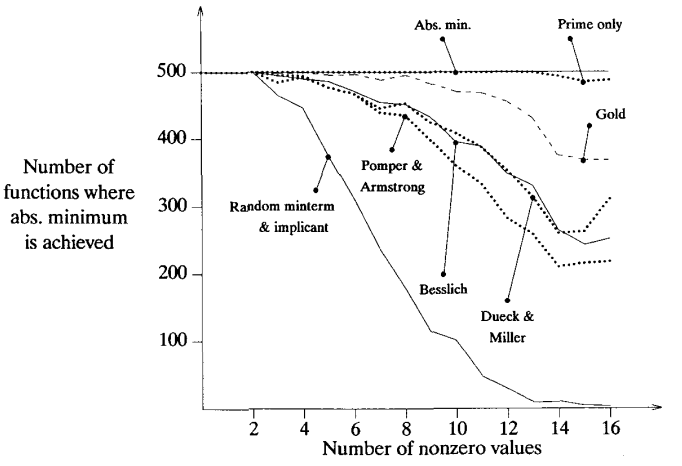


Fig. 6. The number of functions where the absolute minimal realization is achieved versus the number of nonzero values for random symmetric functions.

the programs were run on a time-shared system with time varying load. Our experience, therefore, does not allow us to compare the heuristics on the basis of execution speed. In order to make such a comparison in a fair manner, each individual heuristic must be implemented as efficiently as possible. Also, functions with a larger number of inputs and implicants must be considered to reduce the effect of overhead. The tradeoff is that, for such functions, absolute minimization results may not be obtainable in reasonable time.

V. Absolute Minimization

The program to compute the absolute minimal realization of a function is a search over combinations of implicants. As observed in Section II, it is not sufficient to consider just prime implicants. In the absence of any restricting conditions, we must consider all implicants. In the worst case for 4-valued 2-variable functions ($(f(x_1, x_2) = 3)$, this can be large, 300 implicants. Consider a function which has 100 implicants and requires 8 product terms in a minimal cover. Using a brute force enumeration technique, the proof that this is the minimum, requires that all combinations of seven or fewer implicants be examined. There are more than 17 billion such combinations, and assuming that each combination can be generated and examined in 200 μs (a value that is better than in our program), it would take more than 944 hours, or 39 days, to examine all the combinations for one function! Clearly this is not acceptable.

However, one need not consider all implicants. Specifically, an implicant $I(X)$ of function $f(X)$ can be discarded if it meets one of the following conditions:

1) $I(x) \neq 0 \Rightarrow f(x) = r$ for all assignments x of values to X, i.e., whenever the implicant is nonzero, the function is don't care.
2) $|I(X)| = r - 1$ and $I(X)$ is not a prime implicant.
3) $|I(X)| \neq r - 1$ and there exists another implicant $J(X)$ of $f(X)$ such that $|J(X)| = r - 1$ and $I(X) < J(X)$, where $I(X) < J(X)$ means that for all assignments x of values to X, $I(x) < J(x)$.

Even with this, the search space can be prohibitively large. Using the lemma given below, however, it can be significantly reduced.

Definition: $U(x)$ is a *constrained implicant set* of minterm x for function $f(X)$, if $U(x) = \bigcup_{I(x)>0} I(X)$, where $I(X)$ is an implicant of $f(X)$.

Lemma 1: If $U(x)$ is a constrained implicant set of minterm x for function $f(X)$, then every minimal sum-of-products expression of $f(X)$ where sum is SUM contains at least one implicant from $U(x)$.

Corollary: If there are k constrained implicant sets for a function $f(X)$ each disjoint with all of the other $k - 1$ constrained implicant sets, then no minimal sum-of-products expression of $f(X)$ has fewer than k implicants.

The proof follows from the fact that at least one implicant is needed from each of the k disjoint sets.

Definition: $U(x)$ is a *minimal* constrained implicant set of a function if and only if $0 < |U(x)| \leq |U(y)|$ for all $y \in X$, where $U(x)$ is a constrained implicant set.

A minimal constrained implicant set is useful in an efficient search for the absolute minimal sum-of-products expression for a function. The search space can be represented by a tree where each node corresponds to a function, and each edge to an implicant. The root node represents the input function, and all other nodes represent functions obtained by subtracting from the input function, the implicants in the path (from this node) to the root. If a function has α implicants, the root has α descendents. Furthermore, each of these has at most (and probably less than) $\alpha - 1$ descendents. However, rather than consider all implicants at the root node, it is sufficient to consider only implicants from a minimal constrained implicant set, considerably reducing the search space. From Lemma 1, at least one implicant from this set must be in a minimal sum-of-products expression. For example, the average number of implicants for a 4-valued 2-variable function with one zero, (obtained from a randomly generated sample set of 500 functions) is 111.2, while the average size of the minimal constrained implicant set is only 6.95. This is a considerable reduction. It is also a needed reduction; without it, the computation of the absolute minimum algorithm would not be practical. Interestingly, the average number of prime implicants for the above set is 10.11.

The algorithm recursively finds the absolute minimal cover of the function at each node. Let $\Gamma(f)$ denote the minimum number of implicants needed in the minimal sum-of-products expression of function $f(X)$. Let $U(x) = \{I_1(X), I_2(X), \cdots, I_k(X)\}$ be

any constrained implicant set of $f(X)$. Let $g_i(X) = f(X) - I_i(X)$ be the function obtained by subtracting $I_i(X)$ from $f(X)$. From the definition of a constrained implicant set, it follows that $\Gamma(f) = 1 + \min\{\Gamma(g_1), \Gamma(g_2), \cdots, \Gamma(g_k)\}$.

An outline of the algorithm that finds the minimal sum-of-products expression of a function is as follows.

Absolute Minimization Algorithm

```
**********************************************
algorithm absolute_minimization;

f ← input_function;
cur_best_soln_set ←best solution from the Pomper and Arm-
                   strong, Besslich, and Dueck and Miller
                   heuristics;
cur_best_soln_size ← number of implicants in best solution;
cur_partial_soln_set ← ∅
cur_partial_soln_size ← 0
minimize(f)
stop

**********************************************

procedure minimize (f);

U ← some essential implicant set of f;
                                          [[advance]]
while ((there exists another implicant in U) and
       (cur_partial_soln_size + 1 < cur_best_soln_size)) do
   begin
      I ← the next implicant in U;
      cur_partial_soln_set ← cur_partial_soln_set ∪ {I};
      cur_partial_soln_size ← cur_partial_soln_size + 1
      if (for all assignments x of values to X, f(x) = 0 or
          f(x) = r)
         then
            begin
               cur_best_soln_set ← cur_partial_soln_set;
               cur_best_soln_size ← cur_partial_soln_size
            end;
                                          [[backtrack]]
         else if (cur_partial_soln_size + 1
            < cur_best_soln_size) then
            begin
               minimize (f − I);
            end
         cur_partial_soln_size ← cur_partial_soln_size − 1;
         cur_partial_soln_set ← cur_partial_soln_set − {I}
   end;
return

**********************************************


/*
The subtraction of an implicant I from a function f, as described
by g ← f − I, takes into account the value of the input_function.
Let x be an assignment of values to variables X. Then, g ← f − I
means
   for (all assignments x of values to X) do
      begin
         if ((f(x) = r) or (input_function (x) = r) or
             (f(x) ≤ I(x) and input_function (x) = r − 1))
            then g(x) ← r
            else g(x) ← g(x) − I(x)
      end
*/.
**********************************************
```

The speed of the absolute minimization algorithm depends on the size of the constrained implicant set. Thus, the *smallest* constrained implicant set is desired. However, the time spent searching for the smallest set may be considerable. We choose instead to adopt the following strategy. Choose the present constrained implicant set if its size is less than or equal to some threshold T. Otherwise, examine all constrained implicant sets and choose the smallest. By the choice of T, we control the type of behavior at each node; with a sufficiently large value, the first constrained implicant set is always chosen and with a sufficiently small value, all constrained implicant sets are searched and the smallest chosen. Although reducing the value of T reduces the *total* number of implicants scanned, the extra time required at each node may increase the *overall* program execution time. We tried two values of T, 6 and 9, and found that the former resulted in a somewhat faster program.

The algorithm for finding the minimal sum-of-products expression using only *prime* implicants is a modification of the above. When a cover is found for the input function, a check is made to see if it is a valid solution using only prime implicants. This is done as follows.

1) $g \leftarrow f$ (the input function).
2) If there are no implicants in the solution set of g that are prime with respect to g, stop—this is not a valid solution. Otherwise, remove them from the solution set, subtract them from g, and replace g with the result.
3) If there are no more implicants in the solution set, stop—this is a valid solution. Otherwise, go to step 2.

If the cover is valid, the current best solution set and size are updated, and the algorithm proceeds to look for an improvement over this, as in the previous case. On the other hand, if the cover is not valid, the search continues for a valid cover, as in the previous case. It is interesting that with this algorithm, absolute minimization using only prime implicants is actually *slower* than when all implicants are used. This is because of the additional time spent at each node to verify that the solution consists of prime implicants only. An alternative is to choose from the set of implicants that are prime with respect to the given function. We proceed as follows, select a prime implicant $I_1(X)$ of $f(X)$, subtract $I_1(X)$ from $f(X)$, select a prime implicant of $f(X) - I_1(X)$, subtract $I_2(X)$ from $f(X) - I_1(X)$, etc. The process is the same as in the case of a solution over all implicants involving an exhaustive search except that the choices $I_1(X)$, $I_2(X)$, $\cdots$ are made from all prime implicants. In the latter case, we are guaranteed by Lemma 1 that at least one implicant from *any* constrained implicant set is part of the minimal sum-of-products expression. However, this is *not* the case when the constrained implicant set contains only implicants that are prime with respect to the present function. Suppose, in Fig. 7, we choose the minterm at $x_1 = x_2 = 1$. The constrained implicant set for this minterm consists of only one prime implicant $1^1x_1^{10}x_2^1$. This leads to the sum-of-products expression $1^1x_1^{10}x_2^1 + 1^0x_1^{00}x_2^0 + 1^2x_1^{20}x_2^0$, the second two implicants being prime with respect to the function obtained by the subtraction of $1^1x_1^{10}x_2^1$. However, this is not a minimal solution. That status is held by $1^0x_1^{20}x_2^0 + 1^1x_1^{11}x_2^1$. Therefore, we cannot take advantage of the reduc-

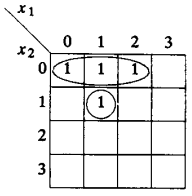


Fig. 7. A 2-variable 4-valued function which requires the use of nonprime implicants to produce a minimal expression.

tion in search offered by the constrained implicant set when only prime implicants are used. Our experience indicates that there is a larger reduction in search time when the search is restricted by constrained implicant sets than by prime implicants. Furthermore, the former guarantees minimality while the latter does not. In the absence of another method to reduce the search, we are faced with the surprising conclusion that prime implicants have less value than constrained implicant sets.

VI. Concluding Remarks

Three heuristic sum-of-products minimization algorithms [3], [6], [14] have been compared on the basis of two sets, 7000 random and 7000 random symmetric 4-valued 2-variable functions. The results from both sets are similar. They show that two heuristics, 1) Dueck and Miller, and 2) Besslich, perform about the same, each slightly better than the 3) Pomper and Armstrong heuristics. All three heuristics perform much better than a heuristic that chooses a solution randomly. Gold, an algorithm that chooses the best of the three heuristics, does reasonably better than any particular one. Our results indicate that, on the average, heuristics perform quite close to optimum. Because of the restriction to two variable functions, we were able to derive the absolute minimal expression for all 7000 functions over each of two sets —random and random symmetric functions. A study of functions with more variables would have precluded this information. The extension of this study to more variables will require either faster computation of the minimal sum-of-products expression or the elimination of this information. Excluding functions where the minimal sum-of-products expression computation is intractable will leave functions, like those with few minterms, where heuristics do reasonably well.

In the process of deriving a tractable absolute minimization algorithm, we made a surprising discovery. Prime implicants have little value in the search for minimal sum-of-products expressions for multiple-valued logic functions where sum is truncated sum. Restricting the search to prime implicants can eliminate minimal solutions. Although such functions are rare, the surprise comes from the fact that for the sets of functions we examined, the use of constrained implicant sets restricts the search *more* than prime implicants *without* compromising minimality.

While the motivation for the study was to compare certain heuristic minimization techniques, we devoted most of the effort to the development of a tractable absolute minimization algorithm. Thus, what began as an investigation of heuristic methods ended as an investigation of methods to reduce the search required by absolute minimization.

Appendix I
Pomper and Armstrong [14] Heuristic Using Interval Literals

```
**************************************

/*
input_function is the given r-valued n-variable function. f is an
intermediate function as it is covered in successive stages of
the algorithm. Each minterm in f can have a value in the range
[0 ··· r] where r is the radix. A value of—
  0 ⇒ this minterm was zero in input_function or was in the
      range [1 ··· (r − 2)] in input_function, but has now been
      fully covered.
  1 ··· (r − 1) ⇒ this minterm still needs to be covered to the
      extent specified by the value
  r ⇒ {this denotes a don't care} this minterm was r in
      input_function or was (r − 1) in input_function, but has
      now been fully covered.
An implicant can be written as I = p·x_1^{j1}·x_1^{i1} ·x_2^{j2}·x_2^{i2} ··· x_n^{jn}·x_n^{in}. By the
size of this implicant, we mean the number of locations where
this implicant is p, which is the range [1 ··· (r − 1)]. That is,
the size_of_implicant_I is

                  ∏ (j_m − i_m + 1).
              1≤m≤n

An implicant J is larger than another implicant I if the size of J
is larger than the size of I.
*/
**************************************

algorithm pomper_and_armstrong;
f ← input_function;
cur_partial_soln_set ← ∅;
while (there exists a minterm in the range [1 ··· (r − 1)]) do
   begin
      α ← a minterm in the range [1 ··· (r − 1)];
      I ← maximal_implicant(α);
      cur_partial_soln_set ← cur_partial_soln_set ∪ {I};
      f ← f − I
   end;
stop

**************************************

function maximal_implicant(α);

cur_size ← − ∞;
cur_terms_covered ← − ∞;
for (each implicant I of f such that I(α) ≥ f(α)) do
   begin
      I terms_covered ← the number of minterms in f that
                would be driven to 0 or r (don't care)
                 if I is subtracted from f;
      I_size ← size_of_implicant_I;
      if (I_terms_covered > cur_terms_covered) or
                ((I_terms_covered = cur_terms_covered) and
                (I_size > cur_size)) then
        begin
           maximal_implicant(α) ← I;
           cur_size ← I_size;
```

```
            cur_terms_covered ← I_terms_covered
        end;
    end;
return
```

* *

```
/*
The subtraction of an implicant I from a function f, as described by f ← f − I, takes into account the value of the input_function. Let x be an assignment of values to variables X. Then, f ← f − I means
        for (all assignments x of values to X) do
            begin
                if ((f(x) = r) or (input_function(x) = r) or
                        (f(x) ≤ I(x) and input_function(x)
                        = r − 1)
                    then f(x) ← r
                    else f(x)  ← f(x) − I(x)
            end
*/.
```

* *

Appendix II
Besslich [3] Heuristic Using Interval Literals

* *

```
algorithm besslich;
/*The definitions of Appendix I apply to the following.*/

f ← input_function;
cur_partial_soln_set ← ∅;
while (f has minterms in the range [1 ··· (r − 1)]) do
    begin
        α ← besslich_most_isolated_minterm(f);
        I ← maximal_implicant(α);
        cur_partial_soln_set ← cur_partial_soln_set ∪ {I};
        f ← f − I
    end;
stop
```

* *

```
function besslich_most_isolated_minterm(f);

/*
Code the function by mapping all the 0's to −1; all the don't care values (r's) to 0; and all other values (1 through r − 1) to 1. Let coded _f be this coded form. Compute the weight transform [3] of coded _f and compute the most isolated minterm as follows:
*/

cur_lowest_wt ← ∞;
α ← 00...00;
for each minterm β in coded _f such that coded _f(β) = 1 do
    begin
        /* compute the weight of this minterm with all other minterms and add */
```

$$\mathrm{wt}(\beta) \leftarrow \sum_{\gamma \in \text{coded\_}f} [\text{coded\_}f(\gamma)^* w(\beta, \gamma)];$$

```
        if (wt(β) < cur_lowest_wt) then
            begin
                α ← β;
                cur_lowest_wt ← wt(β)
            end;
    end;
return
```

* *

```
function w(β, γ);

/*
Let the assignment x of values to the variables x_1, x_2, ..., x_n serve as an index to the value of input_function(x). Specifically, let β = β_1β_2 ··· β_{n−1}β_n and γ = γ_1γ_2 ··· γ_{n−1}γ_n.
*/
```

$$D(\beta, \gamma) \leftarrow \sum_{1 \le m \le n} |\beta_m - \gamma_m|;$$

$$w(\beta, \gamma) = 2^{[n(r-1)-D(\beta,\gamma)]};$$

```
return.
```

* *

Appendix III
Dueck and Miller [6] Heuristic Using Interval Literals

* *

```
algorithm dueck_and_miller;
/*The definitions of Appendix I apply to the following.*/

f ← input_function;
cur_partial_soln_set ← ∅;
while (f has minterms in the range [1 ··· (r − 1)]) do
    begin
        α ← dueck/miller_most_isolated_minterm(f);
        I ← best_implicant(α);
        cur_partial_soln_set ← cur_partial soln_set ∪ I;
        f ← f − I;
    end
stop
```

* *

```
function dueck/miller_most_isolated_minterm(f);

/*
Let min_val be the minimum value of all minterms in f that are in the range 1 ··· (r − 1), i.e., neither 0 nor don't care. For each minterm α in the function with value equal to min_val, compute
```

the clustering factor (CF) as follows

$$\mathrm{CF}(\alpha) = \mathrm{DEA}_\alpha(r-1) + \mathrm{EA}_\alpha$$

where EA_α is the number of minterms adjacent to α with which α can be combined in an interval literal and DEA_α is the number of variables (directions) in which α can be combined with a nonzero number of minterms.

Dueck and Miller define the isolation factor IF as the reciprocal of 1 + the clustering factor above and choose the minterm with the highest IF. The same result can be obtained by choosing the minterm with the smallest CF, avoiding the division. Ties are broken by selecting the first generated. Let α be the minterm thus chosen.

```
*/

α = 00···00;
CF_min ← ∞;
for min_val = 1 to r − 1 do
  if CF_min = +∞ do
    begin
      while (there is a minterm β such that
              f(β) = min_val) do
        begin
          CF(β) = DEA_β + EA_β;
          if CF_min > CF(β) then
            begin
              CF_min ← CF(β);
              α ← β
            end;
        end;
    end;
stop

**********

function best_implicant(α);

cur_rbc ← ∞;
for (every implicant I in f that covers α) do
    begin
      if (rbc(I) < cur_rbc) then
        begin
          best_implicant(α) ← I;
          cur_rbc ← rbc(I);
        end;
    end;
return
**********

function rbc(I);

/*
The calculation of rbc shown here is different from that in [6]
to accommodate the interval literal. Some minor variations were
tried, but the average case results were nearly identical. Of these,
the one that appeared to be the best was chosen.
*/
rbc(I) ← 0;
for (each α in I such that f(α) ≠ r) do
    begin
      for (each variable i = 1 to n) do
        begin
          if ((f(α) − I(α)) or
          (the minterm β immediately following α
            in variable i (if such a β exists) is not in
            I and is such that f(β) = f(α) − I(α)
          or
          (the minterm β immediately preceding α
            in variable i (if such β exists) is not in I
            and is such that f(β) = f(α) − I(α)))
                then

              rbc(I) ← rbc(I) − 1;

          if ((the minterm β immediately preceding
            α in variable i (if such β exists) is not in
            I and is such that f(β) = f(α)) or
          (the minterm β immediately following α
            in variable i (if such β exists) is not in
            I and is such that f(β) = f(α))) then

              rbc(I) ← rbc(I) + 1;

        end;
    end;
return.

**********
```

References

[1] E. A. Bender and J. T. Butler, "On the size of PLA's required to realize binary and multiple-valued functions," *IEEE Trans. Comput.*, pp. 82–98, Jan. 1989.

[2] E. A. Bender, J. T. Butler, and H. G. Kerkhoff, "Comparing the SUM with the MAX for use in four-vaulted PLA's," in *Proc. 15th Int. Symp. Multiple-Valued Logic,* May, 1985, pp. 30–35.

[3] P. W. Besslich, "Heuristic minimization of MVL functions: A direct cover approach," *IEEE Trans. Comput.*, pp. 134–144, Feb. 1986.

[4] R. K. Brayton, G. D. Hachtel, C. T. McMullen and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis.* Boston, MA: Kluwer Academic, 1984.

[5] J. T. Butler and H. G. Kerkhoff, "Analysis of input and output configurations for use in four-valued programmable logic arrays," in *Proc. IEE-E: Comput. Digital Techniques,* July 1987, pp. 168–176.

[6] G. W. Dueck and D. M. Miller, "A direct cover MVL minimization using the truncated sum," in *Proc. 17th Int. Symp. Multiple-Valued Logic,* May 1987, pp. 221–227.

[7] J. F. Gimpel, "A method of producing a Boolean function having an arbitrarily prescribed prime implicant table," *IEEE Trans. Electron. Comput.*, pp. 485–488, June 1965.

[8] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations,* R. E. Miller and J. W. Thatcher, Eds. New York: Plenum, 1972, pp. 85–103.

[9] H. G. Kerkhoff and J. T. Butler, "Design of a high-radix programmable logic array using profiled peristaltic charge-coupled devices," in *Proc. 16th Int. Symp. Multiple-Valued Logic,* May 1986, pp. 100–103.

[10] ——, "A module compiler for the design of high-radix CCD PLA's," *Int. J. Electron,* pp. 797–805, Nov. 1989.

[11] Y. H. Ko, "Design of multiple-valued programmable logic arrays," M.S. Thesis, Naval Postgraduate School, Monterey, CA, Dec. 1988.

[12] H.-L. Kuo and K.-Y. Fang, "The multiple-valued programmable logic array and its application in modular design," in *Proc. Int. Symp. Multiple-Valued Logic,* May 1985, pp. 10–18.

[13] D. M. Miller and J. C. Muzio, "On the minimization of many-valued functions," in *Proc. Int. Symp. Multiple-Valued Logic,* May 1979, pp. 294–299.

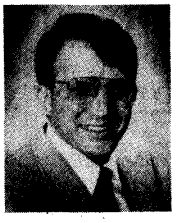
[14] G. Pomper and J. A. Armstrong, "Representation of multivalued functions using the direct cover method," *IEEE Trans. Comput.*, pp. 674–679, Sept. 1981.

[15] J. G. Samson, "Design of a PLA in multiple-valued logic," report "250 uurs opdracht," M.S. Thesis, Univ. of Twente, June 1988.
[16] T. Sasao, *Programmable Logic Arrays: How to Make and How to Use,* ISBN4–526–02026–5 C3054, NIKKAN KOGYOU Pub. Co. 1986 (in Japanese).
[17] T. Sasao, "On the optimal design of multiple-valued PLA's," *IEEE Trans. Comput.,* pp. 582–592, Apr. 1989.
[18] P. P. Tirumalai and J. T. Butler, "On the realization of multiple-valued logic functions using CCD PLA's," in *Proc. 14th Symp. Multiple-Valued Logic,* May 1984, pp. 33–42.
[19] P. P. Tirumalai and J. T. Butler, "Prime and non-prime implicants in the minimization of multiple-valued logic functions," in *Proc. 19th Int. Symp. Multiple-Valued Logic,* May 1989, pp. 272–279.

Parthasarthy P. Tirumalai (M'90) was born in [redacted]. He received the B.Tech degree in electrical engineering from the Indian Institute of Technology, Madras, in 1982 and the M.S. and Ph.D. degrees in computer science from Northwestern University, Evanston, IL, in 1984 and 1989, respectively.

He has been with the Hewlett-Packard Company since January 1985, where he has been involved in the design and diagnosis of logic systems. He is a Hewlett-Packard Resident Fellow (1988) and a Walter P. Murphy Fellow at Northwestern University (1982–1983). His research interests include computer architecture, logic design, and multiple-valued logic.

Dr. Tirumalai received the Siemens India Prize awarded to the best student in the B.Tech E.E. program, IIT-Madras in 1982 and the Ramasarma V. Kolluri Silver Medal awarded to the best student in Electrical Engineering, IIT-Madras in 1981. He received the 14th Prize in the International Student Essay Competition on Electronic and Telecommunication conducted by the United Nations, Geneva in 1974.



Jon T. Butler (S'67–M'67'–SM'82–F'89) [redacted] in Baltimore, MD. He received the B.E.E. and M.Engr. (E.E.) degrees from Rensselaer Polytechnic Institute, Troy, NY, in 1966 and 1967, respectively. He received the Ph.D. degree in electrical engineering from the Ohio State University, Columbus, in 1973.

Since 1987, he has been a Professor in the Department of Electrical and Computer Engineering of the Naval Postgraduate School, Monterey, CA, From 1974 to 1987, he was on the faculty of Northwestern University, Evanston, IL. During that time, he served two periods of leave at the Naval Postgraduate School, first as a National Research Council Senior Postdoctoral Associate (1980–1981) and second as the NAVALEX Chair Professor (1985–1987). From 1973 to 1974, he was a National Research Council Postdoctoral Associate at the Air Force Avionics Laboratory, Wright-Patterson AFB, OH, where he applied cellular automata techniques to pattern processing problems. His research interests include multiple-valued logic and reliable multiprocessing systems. He has made research contributions to cellular automata, binary and multiple-valued logic design, fault-tolerant computing, and software engineering.

Dr. Butler was Chairman of the 1980 International Symposium on Multiple-Valued Logic and was the first Chairman of the Multiple-Valued Logic Technical Committee of the Computer Society from 1980 to 1981. He was an Editor of the IEEE TRANSACTIONS ON COMPUTERS and of the IEEE Computer Society Press. Presently, he is Editor-in-Chief of *Computer*. He was a Co-Guest Editor of a special issue of the IEEE TRANSACTIONS ON COMPUTERS and a Guest Editor of a special issue of *Computer* both on multiple-valued logic. From 1986 to 1988, he has served as Vice-Chair for Hardware of Computer Society's Technical Activities Board. From 1982 to 1985, he was a Distinguished Visitor of the IEEE Computer Society, and presently, he is a member of the Board of Governors of the IEEE Computer Society. He has received the Award of Excellence and the Outstanding Contributed Paper Award for papers presented at the International Symposium on Multiple-Valued Logic.